Programación avanzada en Python

Lección 4: Trabajando con BBDD

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Trabajando con BBDD

PRESENTACIÓN Y OBJETIVOS

En este séptimo capítulo nos adentraremos en el mundo de las BBDD. Primero veremos los fundamentos necesarios del lenguaje de consultas de base de datos SQL y por último como trabajar de forma práctica con SQL y Python mediante SQLite.

Objetivos

En esta lección aprenderás a:

- Trabajar con SQL
- Combinar Python y SQL
- Crear tu propia BBDD

1. SQL

Introducción

SQL es el lenguaje estándar para tratar con bases de datos relacionales. SQL se puede utilizar para insertar, buscar, actualizar y eliminar registros de bases de datos. SQL puede realizar muchas otras operaciones, incluida la optimización y el mantenimiento de bases de datos.

¿Para qué se utiliza SQL?

- Ayuda a los usuarios a acceder a los datos en el sistema RSGBD.
- Ayuda a describir los datos.
- Permite definir los datos en una base de datos y manipular esos datos específicos.
- Con la ayuda de SQL, puede crear y eliminar bases de datos y tablas.
- SQL nos ofrece utilizar la función en una base de datos, crear una vista y un procedimiento almacenado.
- Puede establecer permisos en tablas, procedimientos y vistas.

Una breve historia de SQL

Aquí hay puntos de referencia importantes de la historia de SQL:

- 1970 - El Dr. Edgar F. "Ted" Codd describe un modelo relacional para bases de datos.
- 1974: aparece el lenguaje de consulta estructurado.
- 1978 - IBM lanzó un producto llamado System / R.
- 1986 - IBM desarrolló el prototipo de una base de datos relacional, que está estandarizada por ANSI.
- 1989 - Lanzamiento de la primera versión de SQL
- 1999 - Lanzamiento de SQL 3 con características como disparadores, orientación a objetos, etc.
- SQL 2003: funciones de ventana, características relacionadas con XML, etc.
- SQL 2006: compatibilidad con el lenguaje de consulta XML
- SQL 2011: soporte mejorado para bases de datos temporales

Tipos de declaraciones SQL

A continuación, se muestran cinco tipos de consultas SQL de uso generalizado.

- Lenguaje de definición de datos (DDL)
- Lenguaje de manipulación de datos (DML)
- Lenguaje de control de datos (DCL)
- Lenguaje de control de transacciones (TCL)
- Lenguaje de consulta de datos (DQL)

Lista de comandos SQL

A continuación, se muestra una lista de algunos de los **comandos SQL** más utilizados:

- **Create**: define el esquema de la estructura de la base de datos
- **Insert**: inserta datos en la fila de una tabla
- **Update**: actualiza los datos en una base de datos
- **Delete**: elimina una o más filas de una tabla
- **Select**: selecciona el atributo según la condición descrita por la cláusula WHERE
- **Drop**: elimina tablas y bases de datos

Sintaxis

Todas las declaraciones SQL comienzan con cualquiera de las palabras clave como SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW y todas las declaraciones terminan con un punto y coma (;).

El punto más importante que debemos de tener en cuenta aquí es que SQL no distingue entre mayúsculas y minúsculas, lo que significa que SELECT y select tienen el mismo significado en las declaraciones. Mientras que MySQL hace la diferencia en los nombres de las tablas.

Por lo tanto, si estamos trabajando con MySQL, debe proporcionar los nombres de las tablas tal y como existen en la base de datos.

Principales sintaxis SQL

Instrucción SQL SELECT

```
SELECT column1, column2....columnN
FROM   table_name;
```

Cláusula SQL DISTINCT

```
SELECT DISTINCT column1, column2....columnN
FROM   table_name;
```

Cláusula WHERE de SQL

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION;
```

Cláusula Y / O de SQL

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION-1 {AND|OR} CONDITION-2;
```

Cláusula SQL IN

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name IN (val-1, val-2,...val-N);
```

Cláusula SQL BETWEEN

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name BETWEEN val-1 AND val-2;
```

Cláusula LIKE de SQL

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  column_name LIKE { PATTERN };
```

Cláusula SQL COUNT

```
SELECT COUNT(column_name)
FROM   table_name
WHERE  CONDITION;
```

Cláusula ORDER BY de SQL

```
SELECT column1, column2....columnN
FROM   table_name
WHERE  CONDITION
ORDER BY column_name {ASC|DESC};
```

Cláusula SQL GROUP BY

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name;
```

Cláusula HAVING de SQL

```
SELECT SUM(column_name)
FROM   table_name
WHERE  CONDITION
GROUP BY column_name
HAVING (arithematic function condition);
```

Sentencia SQL CREATE TABLE

```
CREATE TABLE table_name(
column1 datatype,
column2 datatype,
column3 datatype,
.....
columnN datatype,
PRIMARY KEY( one or more columns )
);
```

Sentencia SQL TRUNCATE TABLE

```
TRUNCATE TABLE table_name;
```

Sentencia SQL DROP TABLE

```
DROP TABLE table_name;
```

Sentencia SQL CREATE INDEX

```
CREATE UNIQUE INDEX index_name
ON table_name ( column1, column2,...columnN);
```

Sentencia SQL DROP INDEX

ALTER TABLE table_name
DROP INDEX index_name;
Declaración DESC de SQL
DESC table_name;

Sentencia SQL ALTER TABLE

ALTER TABLE table_name {ADD|DROP|MODIFY} column_name {data_type};

Sentencia SQL ALTER TABLE (Cambiar nombre)

ALTER TABLE table_name RENAME TO new_table_name;

Instrucción SQL INSERT INTO

INSERT INTO table_name(column1, column2....columnN)
VALUES (value1, value2....valueN);

Sentencia SQL UPDATE

UPDATE table_name
SET column1 = value1, column2 = value2....columnN=valueN
[WHERE CONDITION];

Sentencia SQL DELETE

DELETE FROM table_name
WHERE {CONDITION};

Sentencia SQL CREATE DATABASE

CREATE DATABASE database_name;

Sentencia SQL DROP DATABASE

DROP DATABASE database_name;

Sentencia USE SQL

USE database_name;

Sentencia COMMIT de SQL

COMMIT;

Sentencia SQL ROLLBACK

ROLLBACK;

Tipos de datos

El tipo de datos SQL es un atributo que especifica el tipo de datos de cualquier objeto. Cada columna, variable y expresión tiene un tipo de datos relacionado en SQL. Puede utilizar estos tipos de datos al crear sus tablas. Puede elegir un tipo de datos para una columna de tabla según sus necesidades.

SQL Server ofrece seis categorías de tipos de datos para su uso que se enumeran a continuación:

Tipos de datos numéricos exactos

TIPO DE DATOS	DESDE	A
bigint	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
int	-2,147,483,648	2,147,483,647
smallint	-32,768	32,767
tinyint	0	255
bit	0	1
decimal	-10^38 +1	10^38 -1
numeric	-10^38 +1	10^38 -1
money	-922,337,203,685,477.5808	+922,337,203,685,477.5807
smallmoney	-214,748.3648	+214,748.3647

Tipos de datos numéricos aproximados

TIPO DE DATOS	DESDE	A
float	-1.79E + 308	1.79E + 308
real	-3.40E + 38	3.40E + 38

Tipos de datos de fecha y hora

TIPO DE DATOS	DESDE	A
datetime	1 de enero de 1753	31 de diciembre de 9999
smalldatetime	1 de enero de 1900	6 de junio de 2079
date	Almacena una fecha como el 30 de junio de 1991	
hour	Almacena una hora del día como las 12:30 p.m.	

Tipos de datos de cadenas de caracteres

	TIPO DE DATOS y descripción
1	**char** Longitud máxima de 8.000 caracteres (caracteres no Unicode de longitud fija)
2	**varchar** Máximo de 8.000 caracteres (datos no Unicode de longitud variable).
3	**varchar (máximo)** Longitud máxima de 2E + 31 caracteres, datos no Unicode de longitud variable (solo SQL Server 2005).
4	**text** Datos no Unicode de longitud variable con una longitud máxima de 2.147.483.647 caracteres.

Operadores

Un operador es una palabra reservada o un carácter que se utiliza principalmente en la cláusula WHERE de una declaración SQL para realizar operaciones, como comparaciones y operaciones aritméticas. Estos operadores se utilizan para especificar condiciones en una declaración SQL y para servir como conjunciones para múltiples condiciones en una declaración.

- Operadores aritméticos
- Operadores de comparación
- Operadores lógicos
- Operadores utilizados para negar condiciones

Suponga que la **'variable a'** tiene 10 y la **'variable b'** tiene 20.

Operadores aritméticos SQL

Operador	Descripción	Ejemplo
+ (Adición)	Agrega valores a ambos lados del operador.	a + b dará 30
- (Resta)	Resta el operando de la derecha del operando de la izquierda.	a - b dará -10
* (Multiplicación)	Multiplica los valores a ambos lados del operador.	a * b dará 200
/ (División)	Divide el operando de la izquierda por el operando de la derecha.	b / a dará 2
% (Módulo)	Divide el operando de la izquierda por el operando de la derecha y devuelve el resto.	b% a dará 0

Operadores de comparación SQL

Operador	Descripción	Ejemplo
=	Comprueba si los valores de dos operandos son iguales o no, si es así, la condición se convierte en verdadera.	(a = b) no es cierto.
! =	Comprueba si los valores de dos operandos son iguales o no, si los valores no son iguales, la condición se convierte en verdadera.	(a! = b) es cierto.
<>	Comprueba si los valores de dos operandos son iguales o no, si los valores no son iguales, la condición se convierte en verdadera.	(a <> b) es cierto.
>	Comprueba si el valor del operando izquierdo es mayor que el valor del operando derecho, si es así, la condición se convierte en verdadera.	(a> b) no es cierto.
<	Comprueba si el valor del operando izquierdo es menor que el valor del operando derecho, si es así, la condición se convierte en verdadera.	(a <b) es cierto.
> =	Comprueba si el valor del operando izquierdo es mayor o igual que el valor del operando derecho, si es así, la condición se convierte en verdadera.	(a> = b) no es cierto.
<=	Comprueba si el valor del operando izquierdo es menor o igual que el valor del operando derecho, si es así, la condición se convierte en verdadera.	(a <= b) es cierto.
! <	Comprueba si el valor del operando izquierdo no es menor que el valor del operando derecho, si es así, la condición se convierte en verdadera.	(a! <b) es falso.
!>	Comprueba si el valor del operando izquierdo no es mayor que el valor del operando derecho; si es así, la condición se convierte en verdadera.	(a!> b) es cierto.

2. SQLite Python

SQLite es una biblioteca de software que proporciona un sistema de gestión de bases de datos relacional. Lite en SQLite significa ligero en términos de configuración, administración de bases de datos y recursos necesarios.

SQLite tiene las siguientes características: autónomo, sin servidor, sin configuración, transaccional.

Sin servidor

Normalmente, un SGBD como MySQL, PostgreSQL, etc., requiere un proceso de servidor separado para funcionar. Las aplicaciones que quieren acceder al servidor de la base de datos utilizan el protocolo TCP / IP para enviar y recibir solicitudes. Esto se llama arquitectura cliente / servidor. SQLite no requiere un servidor para ejecutarse. La base de datos SQLite está integrada con la aplicación que accede a la base de datos. Las aplicaciones interactúan con la base de datos SQLite para leer y escribir directamente desde los archivos de la base de datos almacenados en el disco.

Autónomo

SQLite es autónomo, lo que significa que requiere un soporte mínimo del sistema operativo o la biblioteca externa. Esto hace que SQLite se pueda utilizar en cualquier entorno, especialmente en dispositivos integrados como iPhones, teléfonos Android, consolas de juegos, reproductores multimedia portátiles, etc.

Sin configuración

Debido a la arquitectura sin servidor, no es necesario "instalar" SQLite antes de usarlo. No hay ningún proceso de servidor que deba configurarse, iniciarse y detenerse. Además, SQLite no utiliza ningún archivo de configuración.

Transaccional

Todas las transacciones en SQLite son totalmente compatibles con ACID. Significa que todas las consultas y cambios son atómicos, consistentes, aislados y duraderos. En otras palabras, todos los cambios dentro de una transacción se llevan a cabo por completo o no se llevan a cabo en absoluto, incluso cuando ocurre una situación inesperada, como un bloqueo de la aplicación, un corte de energía o un bloqueo del sistema operativo.

Creación de una nueva base de datos

Cuando nos conectamos a un archivo de base de datos SQLite que no existe, SQLite crea automáticamente la nueva base de datos.

Para crear una base de datos, primero debemos crear un objeto Connection que represente la base de datos utilizando la función connect() del módulo sqlite3.

Por ejemplo, el siguiente programa de Python crea un nuevo archivo de base de datos pythonsqlite.db en la ruta:

/Users/rsanchezi/Documents/EIP/Master python/PAP/7/

```python
import sqlite3
from sqlite3 import Error


def create_connection(db_file):
    """ create a database connection to a SQLite database """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        print(sqlite3.version)
    except Error as e:
        print(e)
    finally:
        if conn:
            conn.close()


if __name__ == '__main__':
    create_connection(r"/Users/rsanchezi/Documents/EIP/Master python/PAP/7/pythonsqlite.db")
```

En este código:

Primero, definimos una función llamada create_connection() que se conecta a una base de datos SQLite especificada por el archivo de la base de datos db_file. Dentro de la función, llamamos a la función connect() del módulo sqlite3.

La función connect() abre una conexión a una base de datos SQLite. Devuelve un objeto Connection que representa la base de datos. Al utilizar dicho objeto, podemos realizar varias operaciones de base de datos.

En caso de que ocurra un error, lo detectamos dentro del bloque try except y mostramos el mensaje de error. Si todo está bien, se muestra la versión de la base de datos SQLite.

Es una buena práctica de programación que siempre cerremos la conexión de la base de datos cuando terminemos con ella.

En segundo lugar, pasamos la ruta del archivo de la base de datos a la función create_connection() para crear la base de datos. Hay que tener en cuenta que el prefijo r en la ruta indica a Python que estamos pasando una cadena sin formato.

Si omitimos la ruta de la carpeta, el programa creará el archivo de base de datos en el directorio de trabajo actual (CWD).

Si pasamos el nombre del archivo :memory: a la función connect()del módulo sqlite3, creará una nueva base de datos que reside en la memoria (RAM) en lugar de un archivo de base de datos en el disco. El siguiente programa crea una base de datos SQLite en la memoria.

```
def create_connection():
    """ create a database connection to a database that resides
        in the memory
    """
    conn = None;
    try:
        conn = sqlite3.connect(':memory:')
        print(sqlite3.version)
    except Error as e:
        print(e)
    finally:
        if conn:
            conn.close()


if __name__ == '__main__':
    create_connection()
```

Creación de tablas

Para crear una nueva tabla en una base de datos SQLite desde un programa Python, hay que seguir los siguientes pasos:

1. Primero, creamos un objeto Connection usando la función connect() del módulo sqlite3.
2. En segundo lugar, creamos un objeto Cursor llamando al método cursor() del objeto Connection.
3. En tercer lugar, pasamos la declaración CREATE TABLE al método execute() del objeto Cursor y ejecutamos este método.

Veamos un ejemplo:

Primero, desarrollamos una función llamada create_connection() que devuelve un objeto Connection que representa una base de datos SQLite especificada por el parámetro de archivo de base de datos db_file.

```
def create_connection(db_file):
    """ create a database connection to the SQLite database
        specified by db_file
    :param db_file: database file
    :return: Connection object or None
    """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
        return conn
    except Error as e:
        print(e)

    return conn
```

En segundo lugar, desarrollamos una función nombrada create_table() que acepte un objeto Connection y una declaración SQL. Dentro de la función, llamamos al método execute() del objeto Cursor para ejecutar la declaración CREATE TABLE.

```
def create_table(conn, create_table_sql):
    """ create a table from the create_table_sql statement
    :param conn: Connection object
    :param create_table_sql: a CREATE TABLE statement
    :return:
    """
    try:
        c = conn.cursor()
        c.execute(create_table_sql)
    except Error as e:
        print(e)
```

En tercer lugar, creamos una función main() para crear las tablas.

```
def main():
    database = r"/Users/rsanchezi/Documents/EIP/Master python/PAP/7/pythonsqlite.db"

    sql_create_projects_table = """ CREATE TABLE IF NOT EXISTS projects (
                                        id integer PRIMARY KEY,
                                        name text NOT NULL,
                                        begin_date text,
                                        end_date text
                                    ); """

    sql_create_tasks_table = """CREATE TABLE IF NOT EXISTS tasks (
                                    id integer PRIMARY KEY,
                                    name text NOT NULL,
                                    priority integer,
                                    status_id integer NOT NULL,
                                    project_id integer NOT NULL,
                                    begin_date text NOT NULL,
                                    end_date text NOT NULL,
                                    FOREIGN KEY (project_id) REFERENCES projects (id)
                                );"""

    # create a database connection
    conn = create_connection(database)

    # create tables
    if conn is not None:
        # create projects table
        create_table(conn, sql_create_projects_table)

        # create tasks table
        create_table(conn, sql_create_tasks_table)
    else:
        print("Error! cannot create the database connection.")
```

Insertando datos

Para insertar filas en una tabla en la base de datos SQLite, tenemos que seguir los siguientes pasos:

1. Primero, nos conectamos a la base de datos SQLite creando un objeto Connection.
2. En segundo lugar, creamos un objeto Cursor llamando al método cursor del objeto Connection.
3. En tercer lugar, ejecutamos una instrucción INSERT . Si queremos pasar argumentos a la declaración INSERT, utilizamos el signo de interrogación (?) como marcador de posición para cada argumento.

Veamos un ejemplo:

Primero, creamos una nueva función para establecer una conexión de base de datos a una base de datos SQLite especificada por el archivo de base de datos.

```
def create_connection(db_file):
    """ create a database connection to the SQLite database
        specified by db_file
    :param db_file: database file
    :return: Connection object or None
    """
    conn = None
    try:
        conn = sqlite3.connect(db_file)
    except Error as e:
        print(e)

    return conn
```

A continuación, desarrollamos una función para insertar un nuevo registro.

```
def create_project(conn, project):
    """
    Create a new project into the projects table
    :param conn:
    :param project:
    :return: project id
    """
    sql = ''' INSERT INTO projects(name,begin_date,end_date)
              VALUES(?,?,?) '''
    cur = conn.cursor()
    cur.execute(sql, project)
    conn.commit()
    return cur.lastrowid
```

En esta función, usamos el atributo lastrowid del objeto Cursor para recuperar la identificación generada.

Por último, desarrollamos la función main() que llama a la función declarada anteriormente

```
def main():
    database = r"/Users/rsanchezi/Documents/EIP/Master python/PAP/7/pythonsqlite.db"

    # create a database connection
    conn = create_connection(database)
    with conn:
        # create a new project
        project = ('Cool App with SQLite & Python', '2015-01-01', '2015-01-30');
        project_id = create_project(conn, project)
```

Actualizar datos

Para actualizar datos en una tabla desde un programa Python, seguiremos estos pasos:

1. Primero, creamos una conexión de base de datos a la base de datos SQLite usando la función connect(). Una vez creada la conexión a la base de datos, podemos acceder a la base de datos utilizando el objeto Connection.
2. En segundo lugar, creamos un objeto Cursor llamando al método cursor() del objeto Connection.
3. En tercer lugar, ejecutamos la declaración UPDATE llamando al método execute().

A continuación, veremos un ejemplo solo añadiendo las partes específicas de la acción de actualizar ya que las comunes son como las descritas en las secciones anteriores.

```
def update_task(conn, task):
    """
    update priority, begin_date, and end date of a task
    :param conn:
    :param task:
    :return: project id
    """
    sql = ''' UPDATE tasks
              SET priority = ? ,
                  begin_date = ? ,
                  end_date = ?
              WHERE id = ?'''
    cur = conn.cursor()
    cur.execute(sql, task)
    conn.commit()
```

```
def main():
    database = r"/Users/rsanchezi/Documents/EIP/Master python/PAP/7/pythonsqlite.db"

    # create a database connection
    conn = create_connection(database)
    with conn:
        update_task(conn, (2, '2015-01-04', '2015-01-06', 2))


if __name__ == '__main__':
    main()
```

Consultando datos

Para consultar datos en una base de datos SQLite desde Python, seguimos los siguientes pasos:

1. Primero, creamos una conexión a la base de datos SQLite creando un objeto Connection.
2. A continuación, creamos un objeto Cursor utilizando el método de cursor del objeto Connection.
3. Después, ejecutamos una declaración SELECT.
4. Después de esto, llamamos al método fetchall() del objeto cursor para obtener los datos.
5. Finalmente, hacemos un bucle con el cursor para procesar cada fila individualmente.

A continuación, veremos un ejemplo solo añadiendo las partes específicas de la acción de consultar ya que las comunes son como las descritas en las secciones anteriores.

```python
def select_all_tasks(conn):
    """
    Query all rows in the tasks table
    :param conn: the Connection object
    :return:
    """
    cur = conn.cursor()
    cur.execute("SELECT * FROM tasks")

    rows = cur.fetchall()

    for row in rows:
        print(row)
```

En la función select_all_tasks(), creamos un cursor, ejecutamos la instrucción SELECT y llamamos a fetchall() para buscar todas los registros. Veamos otro ejemplo con la cláusula where:

```python
def select_task_by_priority(conn, priority):
    """
    Query tasks by priority
    :param conn: the Connection object
    :param priority:
    :return:
    """
    cur = conn.cursor()
    cur.execute("SELECT * FROM tasks WHERE priority=?", (priority,))

    rows = cur.fetchall()

    for row in rows:
        print(row)
```

Borrando registros

Para eliminar datos en la base de datos SQLite desde un programa Python, utilizamos los siguientes pasos:

1. Primero, creamos una conexión con la base de datos SQLite creando un objeto Connection usando la función connect().
2. En segundo lugar, creamos un objeto cursor
3. En tercer lugar, ejecute la declaración DELETE utilizando el método execute() del objeto Cursor . En caso de que desee pasar los argumentos a la declaración, utilizamos un signo de interrogación (?) para cada argumento.

```python
def delete_task(conn, id):
    """
    Delete a task by task id
    :param conn:  Connection to the SQLite database
    :param id: id of the task
    :return:
    """
    sql = 'DELETE FROM tasks WHERE id=?'
    cur = conn.cursor()
    cur.execute(sql, (id,))
    conn.commit()

def delete_all_tasks(conn):
    """
    Delete all rows in the tasks table
    :param conn: Connection to the SQLite database
    :return:
    """
    sql = 'DELETE FROM tasks'
    cur = conn.cursor()
    cur.execute(sql)
    conn.commit()
```

3. PUNTOS CLAVE

SQL es el lenguaje estándar para tratar con bases de datos relacionales. SQL se puede utilizar para insertar, buscar, actualizar y eliminar registros de **bases de datos**.

Todas las **declaraciones SQL** comienzan con cualquiera de las palabras clave como SELECT, INSERT, UPDATE, DELETE, ALTER, DROP, CREATE, USE, SHOW y todas las declaraciones terminan con un punto y coma (;).

SQLite es una biblioteca de software que proporciona un sistema de gestión de bases de datos relacional. Lite en SQLite significa ligero en términos de configuración, administración de bases de datos y recursos necesarios.